TÉCNICO LISBOA

Evaluation of Real Time Operating System in RISC-V

Rui Silva Ferreira

Electrical and Computer Engineering

Supervisor(s): Prof. Paulo Flores
Prof. Rui Policarpo Duarte

June 2023

Abstract

Real-time operating systems play a critical role in improving an embedded system's performance by efficiently managing multiple tasks. The emergence of the RISC-V instruction set architecture as a standard open-source platform offers unique advantages for embedded systems due to its simplicity and modularity. This thesis will study and adapt an open-source real-time operating system on a RISC-V processor that runs on an FPGA. Given that the RISC-V CPU can be attached to a hardware accelerator, this thesis will also study the interaction between the real-time operating system and the accelerator and how the accelerator impacts the system's performance.

Keywords: RISC-V, RTOS, hardware accelerators

Contents

Chapter 1

Introduction

1.1 Motivation

FIXME Melhorar

A Real-Time operating system (RTOS) is one of the components that support the execution of applications in a computational system that can improve the performance of that system. RTOS are often designed to meet the timing requirements of real-time systems, therefore, its use is suitable to manage multiple tasks with varying priorities. An RTOS also provides inter-task communication, mutual exclusion, task synchronization, and power modes. These capabilities are crucial for the system's functioning and to manage the system's resources.

Recently, RISC-V has been establishing itself as a standard open-source instruction set architecture (ISA) both in the academic environment as well as in industry implementations [1]. The open-source nature of RISC-V enables a broader and larger community around it that contributes to its development. Its modular and simplistic approach makes it not only easier but also more suitable for systems with resource constraints.

Hardware accelerators are used in embedded systems by adding custom digital circuits to speed-up algorithms that usually are the computational bottleneck. When using hardware accelerators to perform computationally intensive tasks for which they were specifically designed the system is able to better utilize its resources. The use of accelerators can also improve the responsiveness of the system to the time constraints of real-time systems.

In addition to the performance increase that a hardware accelerator provides, the use of an RTOS in such systems will help take advantage of the CPU time that would otherwise be wasted. Using RISC-V based processor will allow the developer to implement a system that is specific for its application, optimizing, even more, the resource utilization of the system.

1.2 Objectives

FIXME Mudar os objetivos para refletir melhor a tese

The primary objective of this thesis is to incorporate a RTOS into a RISC-V processor inside an SoC programmed in a field programmable gate array (FPGA), and use the RTOS capabilities to manage and use a hardware accelerator. The initial work is to survey the existing SoC based on RISC-V implementations and RTOS. After choosing a RISC-V core and system on chip (SoC), they will be targeted on an FPGA. The third stage is to implement and evaluate how the RTOS performs in the RISC-V Soc. The following action is to include a hardware accelerator in the system. Finally, the last stage is to evaluate the performance of the system and to observe how the use of an accelerator will impact the performance of the RTOS.

1.3 Document Outline

TODO Document Outline para o projeto final

Chapter 2

Background

2.1 RISC-V Instruction Set Architecture

RISC-V [2], [3] is an instruction set architecture (ISA) that is rapidly growing due to the fact that it is an open-source ISA, contrary to other popular ISAs such as ARM's and Intel's x86. The RISC-V ISA is defined to be a base ISA that is restricted to a minimal set of instructions capable of providing a target to compilers, assemblers, operating systems, etc. In addition to the base ISAs, RISC-V International [4] also provides standard extensions and the capability to add custom extensions that enable the developer to add additional capabilities for specific applications.

There are four base ISAs that are characterized by the width of the integer registers and by the number of integer registers. The two primary base ISAs are RV32I and RV64I which provide 32-bit and 64-bit address space, respectively. Both have 32 integer registers. The RV32E is a subset of the RV32I that only has 16 registers and was created to support small microcontrollers. The fourth base ISA is RV128I and supports a 128-bit address space. As mentioned before, RISC-V International provides a few extensions that add some functionalities to the base instruction sets. The most significant among them is the 'M' standard extension that adds integer multiplication and division for signed and unsigned integers. Other extensions are the 'A' standard extension that adds atomic instructions, the 'F' standard extension that provides floating-point operations, floating-point status and control registers and floating-point registers and the 'Zicsr' extension that allows access to the control and status register, this enables interrupt and exception handling.

RISC-V also supports privileged levels and extensions for the privileged ISAs. There are currently three privilege levels: User (U), Supervisor (S) and Machine (M). The privilege modes exist to add protection between components of the software stack. The Machine mode has the most privileges and runs implementation specific firmware. The Supervisor mode is typically used by operating systems and it gives access to most of the hardware. The mode that offers the least privileges is the User mode. It's in this mode that user applications generally run. All implementations must provide the Machine mode since this is the only privileged mode that has access to the entire hardware.

2.1.1 RISC-V Hardware Implementations

There are many hardware implementations of RISC-V, however, for the purposes of this work, the only implementations analyzed were the ones listed on the list [5] under the SoC platforms section, since the focus of this thesis is not to develop the SoC itself. Even though there are several implementations listed it's necessary that the chosen SoC is under an open source license, has good documentation, and an active community support.

Three candidate RISC-V implementations were selected to be studied in detail: LiteX [6], SweRVolf [7] and NEORV32 [8]. Table 2.1 presents a comparison of the main key factors, such as the base ISA, the supported extensions, the provided RAM size, the system clock speed, the supported simulation tool, and the quality of the documentation provided.

	LiteX	SweRVolf	NEORV32
Base ISA	RV32I	RV32I	RV32I
RISC-V Extensions	[M][A][F][D][C]	[M][C]	[M][C][Zicsr][Zicntr]
RAM	256MB	128MB	configurable
system clock	1MHz	50MHz	150MHz
Simulation tool	Verilator	Verilator	ISIM
Support for RTOS	Zephyr	Zephyr, TockOS	Zephyr, FreeRTOS
Documentation	decent	good	decent

Table 2.1: Comparison between RISC-V SoC

Looking at table 2.1 it's possible to see that the NEORV32 SoC is capable of higher clock speeds and has support for two of more relevant RTOS as will be discussed in Section 2.2. However, none of these features is enough to make it the obvious choice, since it also is the more complex one. The LiteX SoC has native support for the Zephyr OS, having a guide on how to install and simulate the RTOS on the Zephyr documentation. The SweRVolf SoC has good support and documentation and already supports relevant RTOS, Zephyr and TockOS. SweRVolf SoC is the SoC selected for this thesis work as it offers resources for the integration of the RTOS that reduces the development effort and ensures a correct integration of the chosen RTOS.

2.2 Real-Time Operating Systems

A Real-time System is a system where its correctness does not depend only on its output but also depends on the time that the output takes to produce. These systems can be divided into three types: hard real-time systems; soft real-time systems; and firm real-time systems. Hard RTS are those where the system will fail if any task does not finish in its set time. In soft RTS, tasks can fail their time constraints, however, the performance of the overall system can be reduced because of this. Firm RTS have tasks that need to be processed before their deadline, but having some tasks fail that deadline will not make the system fail.

In more complex cases, the use of Operating Systems (OS) is necessary to manage the resources to meet the demands of the system. A Real-time Operating System is an OS designed to take into consideration the necessity of RTS to comply with time constraints. Since an RTOS is an OS it must have some basic functionalities in its kernel such as a process scheduler, memory management, a file system, tasks, multitasking, synchronization, timers and clocks, inter task communication, I/O, and memory management.

One of the most important capabilities of an OS is its capability to multitask. Since only one task per CPU core can run at any given time, the OS can swap between tasks to improve the system's performance. In those cases, the OS saves the context of the initial task and starts running the highest priority task. Multitasking is especially important in RTOS. When a high priority task is ready to execute it is imperative that the RTOS is able to execute that task so that no time constraint is broken.

In typical RTOS, tasks are grouped into three states: running, ready, and blocked. The running task is the task that is executing on the CPU. Ready tasks are those tasks that are ready to run and are waiting to go to the running state. Finally, the blocked state includes the tasks that cannot run at that moment, usually because they are waiting for an event to occur. When there are multiple tasks ready to be executed the kernel must decide which task is going to use the available resources. In this case, the kernel uses a scheduler. The scheduler is the process that controls which task is running on the CPU at any given time. To execute the tasks, the scheduler makes use of threads. Threads are parts of processes that can run independently of other threads since they have their own stack and registers copy. Threads within the same process also share some resources such as memory and files.

In RTOS, several tasks have to wait for events provided either by other tasks or by the hardware in the system. These events are called interrupts. To handle the interrupts, the RTOS makes use of a mechanism called interrupt service routine (ISR).

2.2.1 Task Scheduling and Context Switching

The scheduler is the mechanism that is in charge of choosing the task that is executed at any time. The scheduler is especially important in RTOS since it has to ensure that all tasks meet their time constraints. Scheduling algorithms can be divided into two types, preemptive, where the scheduler has the capability to switch the running task even if that task does not go into the blocked state, and cooperative, where the scheduler does not swap the task that is in the running state, but instead, the running task yields the resources periodically. There are a few situations where the scheduler needs to decide what task should run next: when the running task finishes, when the running task moves to the blocked state or when an interrupt happens. In RTOS the most commonly used scheduling algorithms are preemptive, since they provide more control over what task is in the running state.

There are a number of preemptive scheduling algorithms. The most commonly used is priority scheduling or highest priority first. In this scheduling algorithm, every task is assigned a priority. The scheduler then makes sure that the task with the highest priority is always running. If a low priority task is running and a new high priority task goes to the ready state, then that new task starts running and

the old task goes to the ready state. Another scheduling algorithms are first come first served, or FIFO, where the selected task is the first task to arrive at the ready state, and shortest job first, where the task with the least amount of computing left executes first. One example of a cooperative scheduling algorithm is round-robin scheduling. With this algorithm the tasks are not assigned any priority, however, the tasks are swapped after a set period of time. It's very common for RTOS to use this type of scheduler to assign running time to tasks of the same priority in priority scheduling.

Another important mechanism of the RTOS is how it handles context switching. Context switching is the mechanism that saves the current state of the task for later use, usually to its stack, and initializes the state of the new running task. This adds overhead that, for RTSs, can make the system fail. That is why it's important that the context switching in RTOS takes as little time as possible. The frequency of context switches depends on the scheduling policy and preemptive scheduling, usually, has more context switches than non-preemptive schedulers. This means that the scheduler has to weigh the context switching delay when choosing what task to execute.

2.2.2 Interrupt Handling

One of the ways that tasks go from the blocked state to the ready state is when some event occurs. This event can come in the form of software interrupts, such as messages from other tasks, and hardware interrupts, such as input from a keyboard or mouse. To handle the interrupts there is a mechanism called Interrupt Service Routine (ISR). The ISR is a high priority task that runs every time an interrupt occurs and manages the cause of that interrupt. After the triggering of an interrupt, an interrupt request (IRQ) that is associated with the interrupt signals the ISR. The ISR then utilizes an interrupt vector table to determine the correct interrupt handler function. All of this adds a delay which is called interrupt latency. More precisely, interrupt latency is the time between the interrupt signal and the conclusion of the ISR.

Before starting the ISR, the CPU has to save the context of the task it was executing. And after completing the interrupt handling routine the CPU can restore the state it was on or start a new task that was triggered by the interrupt. Sometimes the running thread may need to prevent ISRs from executing, due to some time-sensitive task or critical section operation. In those cases it is possible to disable the ISR, however, this should only be done when no other solution is available, since a more critical task may be delayed because of it.

2.2.3 Resource Sharing

Since the purpose of the RTOS is to manage and run multiple processes, it's important to mention how the RTOS shares the data and the resources used between the tasks. It's generally unsafe for systems capable of multitasking to access the same data without some kind of protection. Protection may be achieved using the following methods.

Temporarily Masking

Temporarily masking is when the user temporarily disables interrupts and the ability for the kernel to switch tasks. This way the task can process the data knowing that no other tasks will change that data before the processing finishes. Temporarily masking is the fastest way to make sure that the current task has exclusive access to the systems resources, however it should only be used when the longest path through the protected section is smaller then the interrupt latency since this will prevent any other task from running, even higher priority ones.

Semaphores and Mutexes

A semaphore is an object that controls the access to a specific resource. The semaphore stores how many tasks can access that resource and how many tasks are accessing it at that time. When the number of tasks reaches the limit, no other tasks are allowed to access that resource. This way whenever a task needs to access shared resources it increments the semaphore counter. When that thread finishes it's processing it decreases the semaphore counter. When only one task can access that resource the semaphore is called a binary semaphore.

Similarly to binary semaphores, there is a mechanism called mutex. A mutex ensures mutually exclusive access to a hardware or software resource. When a task wants to access a mutually exclusive resource it must start by locking the associated mutex. After the task is finished it unlocks the locked mutex. While the mutex is locked no other tasks can execute because they cannot access the resources protected by the mutex. The task only starts executing again when the mutex is unlocked. One key difference between mutexes and semaphores is priority inversion. Priority inversion happens when a low priority task locks a mutex and a high priority task is blocked by that mutex. When this happens the priority level of the original task is elevated so that the higher priority task does not have to wait for the low priority task to be assigned CPU time. One problem that comes with mutexes is deadlocking. This happens when, for example, task A locks task B, however, before locking, task B also locked task A. When this happens both tasks are unable to continue executing. This can be prevented by carefully designing the program.

Message Passing

Message passing mechanisms such as FIFOs, stacks, message queues and a mailbox can be used to share data between tasks without the use of shared memory. With this mechanisms a task only shares the data after its processing is done, ensuring that the other tasks only start processing after receiving the data from the first task. Message passing can also make use of priority inversion when a high priority task needs data from a low priority task. Deadlock can also occur when some task is working on data needed for another task, but needs data from the blocked task to continue.

2.2.4 RTOS Comparison

Before starting the incorporation of the accelerator into the RTOS it's necessary to choose a reliable RTOS. A series of open source RTOS, seen in [9], were analyzed. Immediately, several of the RTOS in the list were excluded because they either stopped being updated or because they lacked good documentation. After this initial filtering, the RTOS that stood out from the rest were: FreeRTOS [10], RIOT [11] and Zephyr [12].

FreeRTOS

FreeRTOS is an RTOS that targets small embedded systems. The RTOS has a minimal footprint and prioritizes its small memory size, low overhead, and fast execution. The RTOS Kernel can be tailored to the specific use case of the system with a configuration file called FreeRTOSConfig.h. In addition, its small and simple kernel makes it ideal for systems with limited resources.

Besides the expected capabilities of an RTOS, FreeRTOS also provides some more specific ones. First, it has a very small footprint needing only 4.4 KB of ROM and 500 Bytes of RAM. FreeRTOS has a very fast context switch delay, needing only 84 cycles to swap between tasks. The RTOS uses a highest priority scheduling algorithm, with Round Robin time slicing. Both of these algorithms can be disabled in the configuration file. On top of all this, FreeRTOS also provides excellent documentation and vast community support.

RIOT

RIOT is an RTOS designed to cater to the requirements of Internet of Things devices. The main goal of RIOT is to provide a lightweight but versatile RTOS that runs on a low-power microcontroller but can also use the full power of larger devices with more resources. To achieve this, RIOT uses a modular architecture that allows developers to include only the components needed for the requirements of the system. The programming model is based on event-driven programming. This allows for a highly responsive and efficient system, that still supports multi-tasking. RIOT has a very small footprint requiring only 3.2 KB of ROM and 2.8 KB of RAM. It also has a very small interrupt latency that takes only 50 clock cycles. The RTOS uses a highest priority first scheduling algorithm and a first come first served algorithm for tasks with the same priority.

Zephyr

Zephyr is an RTOS developed by the Linux Foundation. It offers a scalable and secure platform for IoT devices. Similarly to RIOT, it offers a modular architecture that allows for developers to only select the necessary features of the RTOS. This RTOS provides developers a feature rich software that can run in a footprint as small as 8kB and as large as to reach megabytes and supports both 32bit and 64bit architectures.

Besides the expected capabilities of an RTOS, Zephyr offers additional capabilities. First, it has a small footprint, requiring 8 KB of ROM and 4 KB of RAM. It also has a fast context switch delay, taking

264 clock cycles. This RTOS uses a highest priority first scheduler and utilizes the algorithm longest waiting first for tasks of equal priority. It also allows the developer to choose if RTOS will use any time slicing or not. It is also possible to define the task queuing strategies used in the scheduler.

Feature Comparison

There are a few characteristics to compare between the RTOS. First, it is important to make sure that the selected RTOS is supported in the chosen SoC, and if not, if the board fits the installation requirements, such as ROM and RAM. It is also important to study how fast can the RTOS process an interrupt and how fast can it switch its context. Finally, it's important to choose an RTOS that has good documentation and strong community support. Table 2.2 summarizes the characteristics mentioned. Note that some of the more precise metrics, such as interrupt latency and context switch delay, do not have any strict value since they depend on the use case of the RTOS.

	FreeRTOS	RIOT	Zephyr
Minimum ROM size	4.4 KB	3.2 KB	8 KB
Minimum RAM size	500 Bytes	2.8 KB	5 KB
Interrupt Latency	—	50 cycles	—
Context Switch Delay	84 cycles	—	264 cycles
Scheduling Policy	Highest Priority	Highest Priority	Highest Priority
Time Slicing Policy	Round Robin	none	configurable
Documentation	Excellent	Good	Excellent

Table 2.2: Comparison between FreeRTOS, RIOT and Zephyr

Looking at the metrics mentioned in Table 2.2, t it's possible to see that FreeRTOS should perform better for smaller systems that have less RAM and ROM available. It is also a better RTOS when dealing with small tasks and the context switch delay is more impactful. At the same time, due to its vast variety of capabilities, it's possible to conclude that ZephyrOS should perform better for larger systems. In this system the context switch delay should not be as impactful, making the pros of the RTOS outweigh the cons. The support for the chosen SoC is also an important feature to consider. None of the RTOS analyzed have direct support in their documentation page, however, both FreeRTOS and Zephyr have online guides on how to use them with the chosen SoC, SweRVolfX.

Since ZephyrOS is the RTOS with better support and documentation it will be the one used. However, it's important to keep in mind that, if the capabilities provided are not useful or even worsen the performance of the overall system, the lighter option should be used.

Chapter 3

RVfpga

3.1 Baseline SoC: RVfpgaNexys

In section 2.1, after analyzing several different RISC-V based SoCs, the SweRVolf SoC was chosen as the baseline SoC for this thesis. RVfpgaNexys is an extended version of SweRVolf targeted to the Nexys A7 board. The extended SweRVolf, or SweRVolfX, adds 4 peripherals to SweRVolf: a GPIO interface for the board's LEDs and switches, a PTC, an additional SPI, and a controller for the 8 digit 7-Segment Displays.

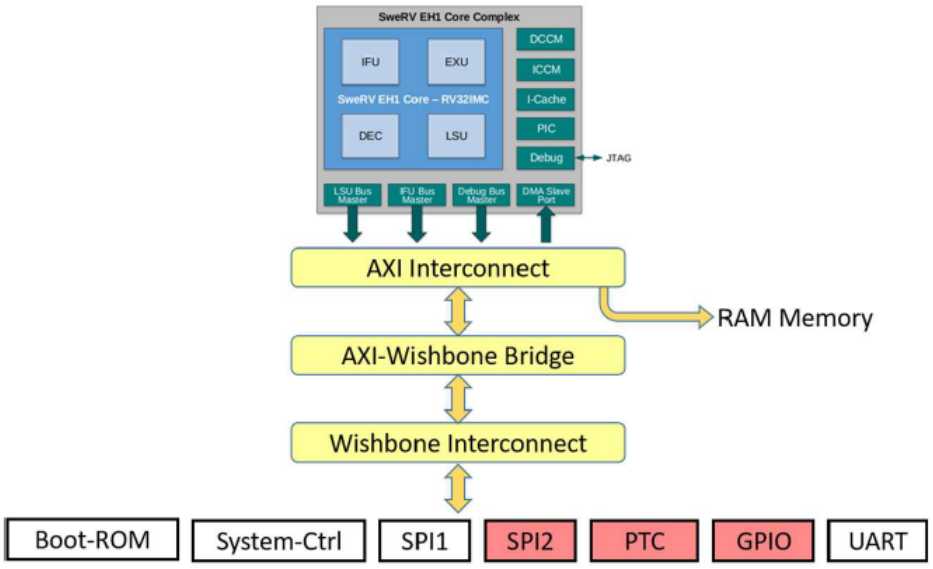

Figure 3.1: SweRVolfX block diagram from Imagination Technologies

Figure 3.1 shows the block diagram of the SweRVolfX SoC. The SoC utilizes the SweRV EH1 Core Complex from Western DIgital with a 32-bit (RV32ICM) core. The SweRVolf SoC includes a Boot ROM, UART, a System Controller and an SPI controller. The peripherals use a Wishbone bus, however the SweRV EH1 Core uses AXI, so the SoC also includes a AXI to Wishbone bridge. The RAM memory is connected to the processor through the AXI interconnect.

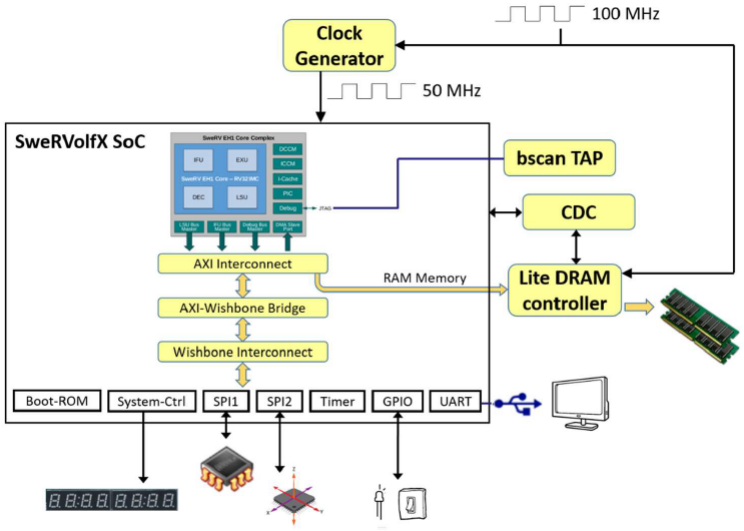

Figure 3.2: RVfpgaNexys block diagram from Imagination Technologies

Figure 3.2 has a block diagram that shows how the SweRVolfX SoC and the Nexys A7 board interact with each other.

TODO Recursos utilizados pela SoC TODO Benchmarks corridos

3.2 AXI Interconnect

TODO Explicar AXI Interconnect TODO O que é preciso adicionar? TODO Primeiro tentei modificar o interconnect que já lá estava. Porque não funcionou? TODO Depois tentei utilizar uma opção open source para manter a SoC completamente simulável e open source. Porque é que não deu? TODO Finalmente utilizei o AXI interconnect da xilinx. Porque foi a melhor opção? TODO Quais são os ports que lá estão e para que servem? TODO Recursos utilizados pela SoC e comparar com a versão anterior TODO Benchmarks corridos e comparar com a versão anterior

3.3 Hardware Accelerator

TODO Explicar acelerador TODO Qual é o funcionamento básico do acelerador? TODO Como o desenhei no HLS? Quais foram os recursos utilizados para o acelerador? TODO Como é que a SoC interaje com o acelerador? TODO Recursos utilizados pela SoC e comparar com a versão anterior TODO Benchmarks corridos e comparar com a versão anterior

Chapter 4

Software Developed

Chapter 5

RTOS in RISC-V

Chapter 6

Conclusions

6.1 Future Work

References

[1] Samuel Greengard. Will risc-v revolutionize computing? *Commun. ACM*, 63(5):30–32, apr 2020. ISSN 0001-0782. doi: 10.1145/3386377. URL https://doi.org/10.1145/3386377.

[2] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V International, 20191213 edition, December 2019.

[3] Krste Asanovic Andrew Waterman and John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International, 20211203 edition, December 2021.

[4] Risc-v international. https://riscv.org/, . Accessed: 2023-04-15.

[5] Risc-v cores and soc overview. https://github.com/riscvarchive/riscv-cores-list, . Accessed: 2023-04-15.

[6] Litex documentation. https://github.com/enjoy-digital/litex. Accessed: 2023-05-06.

[7] Swervolf documentation. https://github.com/chipsalliance/Cores-SweRVolf. Accessed: 2023-05-06.

[8] The neorv32 risc-v processor. https://github.com/stnolting/neorv32. Accessed: 2023-05-06.

[9] List of open source real-time operating systems. https://www.osrtos.com/. Accessed: 2023-05-28.

[10] Freertos: Real-time operating system for microcontrollers. https://freertos.org/. Accessed: 2023-05-28.

[11] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6): 4428–4440, 2018. doi: 10.1109/JIOT.2018.2815038.

[12] Zephyr project, a linux foundation project. https://www.zephyrproject.org/. Accessed: 2023-05-28.